1. **What are the new features you have worked on Java 8?**

   - Lambda Expressions
   - Functional Interfaces
   - Static and Default methods in Interface.
   - Method Reference
   - Streams
   - Predefined Functional Interfaces
   - Local Date & Time API.
   - Optional class
   - forEach
   - StringJoiner

2. **What are main advantages of using Java 8?**

   - ✓ Less Boilerplate of code
   - ✓ More readable and reusable code
   - ✓ More testable code
   - ✓ Parallel operation

3. **Explain about Lambda Expressions and its advantages?**

   - ✓ In Java8 Lambda Expressions came to introduced the functional programming.
   - ✓ Lambda expression is an anonymous function.
   - ✓ It provides a clear and concise way to implement the Functional Interfaces.
   - ✓ It is very useful in collection library. It helps to iterate, filter and extract data from collection.

   **Advantages:**

   **----------**

   - ✓ Functional Programming.
   - ✓ Java lambda expression is treated as a function, so compiler does not create .class file.
   - ✓ It provides a clear and concise way to implement SAM interface (Single Abstract Method) by using an expression. It is very useful in collection library in which it helps to iterate, filter and extract data.

   **Lambda Expression Syntax:**

   **----------**

   **(argument-list) -> {body}**

1) **Argument-list:** It can be empty or non-empty as well.

2) **Arrow-token:** It is used to link arguments-list and body of expression.

3) **Body:** It contains expressions and statements for lambda expression.

Lambda Expressions Examples with multiple arguments:

**Zero parameter:** () -> System.out.println("Zero parameter lambda");

**One parameter:** (p) -> System.out.println("One parameter: " + p);

**Multiple parameters:** (p1, p2) -> System.out.println("Multiple parameters: " + p1 + ", " + p2);

Example: Zero parameters with and without Lambda Expressions

```
interface Test{
    void display();
}

public class LamdaExpressions {

    public static void main(String[] args) {

        //Without Lamda Expression
        Test test = new Test() {
            @Override
            public void display() {
                System.out.println("In Without Lambda"); //In Without Lambda
            }
        };
        test.display();

        //With Lambda Expressions
         Test test1 = () ->System.out.println("In With Lambda"); //In With Lambda

         test1.display();
    }
```

Example: Two parameters with and without Lambda Expressions

```
interface Test {
    void display(int i, int j,int k);
}

public class LamdaExpressions {

    public static void main(String[] args) {

        // Without Lambda Expression
        Test test = new Test() {

            @Override
            public void display(int i, int j,int k) {
                System.out.println(i+j+k  + " With out Lambda ");//45 With out Lambda

            }
        };
        test.display(10, 15,20);

        // With Lambda Expressions
        Test test1 = (i, j,k) -> System.out.println(i+j+k + " In With Lambda "); //64 In With Lambda

        test1.display(10, 20,34);
```

Example: Invoking the thread with and without Lambda Expressions

```
//With out Lambda Expression
new Thread(new Runnable() {
    @Override
    public void run() {

        System.out.println("Starting thread in old way");
    }
}).start(); //Starting thread in old way

//With Lambda Expression
new Thread(() -> System.out.println("In Lambda Expression way")).start();//In Lambda Expression way
```

Example: For Each with and Without Lambda Expressions

```
public static void main(String[] args) {

    List<String> pointList = new ArrayList<>();
    pointList.add("1");
    pointList.add("2");

    //Without Lamda Expression
    for(String str: pointList) {
        System.out.println(str); //1 2
    }

    //With Lambda Expressions
    pointList.forEach(item -> System.out.println(item)); //1 2
}
```

**4. Explain the rules of Lambda Expression?**

- ✓ A lambda expression can have zero, one or more parameters.
- ✓ The body of the lambda expressions can contain zero, one or more statements.
- ✓ If body of lambda expression has single statement curly brackets are not mandatory and the return type of the anonymous function is the same as that of the body expression.

When there is more than one statement in body than these must be enclosed in curly brackets.

5. **Difference between Lambda Expressions and Anonymous class?**

| **Anonymous Class** | **Lambda Expression** |
|---|---|
| Anonymous class is an inner class without a name, which means that we can declare and instantiate class at the same time. | A lambda expression is a short form for writing an anonymous class. By using a lambda expression, we can declare methods without any name. |
| An anonymous class object generates a separate class file after compilation | It does not generate any .class file. lambda expression is converted into a private method |
| Anonymous classes can be used in case of more than one abstract method | Lambda expression specifically used for **functional interfaces**. |
| We need to provide the function body only in lambda expression | In the case of an anonymous class, we need to write the redundant class definition. |

6. **Explain the advantages of Functional programming?**

- ✓ Functional programming is not a mathematics it is just an abstraction to solve real-world complex problems in an easy and effective manner.
- ✓ Reduce the code redundancy
- ✓ Improves the modularity
- ✓ Supports parallel programming

7. **Differences between Functional and Object oriented Programming?**

| **Functional Programming** | **Object Oriented Programming** |
|---|---|
| This programming paradigm emphasizes on the use of functions where each function performs a specific task. | This programming paradigm is based on object oriented concept. Classes are used where instance of objects are created |
| Fundamental elements used are variables and functions. The data in the functions are immutable | Fundamental elements used are objects and methods and the data used here are mutable data. |
| It uses recursion for iteration. | It uses loops for iteration. |
| It is parallel programming supported. | It does not support parallel programming. |
| Does not have any access specifier. | Has three access specifiers namely, Public, Private and Protected. |
| No data hiding is possible. Hence, Security is not possible. | Provides data hiding. Hence, secured programs are possible. |

8. **Why Do Local Variables Used in Lambdas Have to Be Final or Effectively Final?**

Java 8 has new concept called “Effectively final” variable. It means that a non-final local variable whose value never changes after initialization is called “Effectively Final”. This concept was introduced because prior to Java 8 we could not use a non-final local variable in an anonymous class. If you have access to a local variable in Anonymous class, you have to make it final.

When Lambdas was introduced, this restriction was eased. Hence to the need to make local variable final if it’s not changed once it is initialized as Lambda in itself is nothing but an anonymous class. Java 8 realized the pain of declaring local variable final every time developer used Lambda and introduced this concept and made it unnecessary to make local variables final. So if you see the rule for anonymous class still not changed, it’s just you don’t have to write final keyword every time when using lambdas.

**9. What is Functional Interface?**

- ✓ Interface with only one abstract method is known as Functional Interface. It can have any number of default and static methods. It can also declare methods of object class.
- ✓ A Java functional interface can be implemented by a Java Lambda Expression.
- ✓ Functional Interface is also known as Single Abstract Method Interfaces or SAM Interfaces. It is a new feature in Java, which helps to achieve functional programming approach.
- ✓ @FunctionalInterface annotation is used to ensure that the functional interface can’t have more than one abstract method. In case more than one abstract methods are present, the compiler flags an ‘Unexpected @FunctionalInterface annotation’ message. However, it is not mandatory to use this annotation.
- ✓ A Java lambda expression implements a single method from a Java interface. In order to know what method the lambda expression implements, the interface can only contain a single unimplemented method. In other words, the interface must be a Java functional interface.

  **Predefined functional interfaces in Java:**

  - Runnable
  - Comparable

Example: Basic Functional Interface

```java
@FunctionalInterface
interface Functional{
    public String getName(String name);
}
```

Example: Functional Interface with more default and static methods

```java
@FunctionalInterface
interface Functional{
    public String getName(String name);

    default void show() {
        System.out.println("In defult method 1");
    }

    default void print() {
        System.out.println("Indefault print method");
    }

    static void display() {
        System.out.println("In static method display");
    }

    static void guide() {
        System.out.println("In static method guide");
    }

}
```

Example: We should not have more than abstract method in Functional Interface

```java
@FunctionalInterface
interface Login{ //Invalid '@FunctionalInterface' annotation; Login is not a functional interface
    public void execute();
    public void show();
}
```

Example: Implementing the Functional Interface Methods

```java
@FunctionalInterface
interface Functional{
    public String getName(String name);

    default void show() {
        System.out.println("In defult method 1");
    }

    default void print() {
        System.out.println("Indefault print method");
    }

    static void display() {
        System.out.println("In static method display");
    }

    static void guide() {
        System.out.println("In static method guide");
    }

}

public class FunctionalInterfaceExampe {
    public static void main(String[] args) {
        Functional functional = name -> name;
        System.out.println(functional.getName("test")); //test
    }

}
```

**10. Can we override the public methods of Object classes in Functional Interface?**

If an interface declares an abstract method overriding one of the public methods of java.lang.Object, that also does not count toward the interface's abstract method count since any implementation of the interface will have an implementation from java.lang.Object

```java
@FunctionalInterface
interface Functional{
    public String getName(String name);

    @Override
    int hashCode();

    @Override
    String toString();

}

public class FunctionalInterfaceExampe {
    public static void main(String[] args) {
        Functional functional = name -> name;
        System.out.println(functional.getName("test")); //test

    }

}
```

**11. Why can't default methods override equals, hashCode, and toString?**

First of all will get the syntax error if we define the equals, hashcode and toString as default methods in interface. In general Object class methods required in classes to provide some functionality not in interfaces.

```
interface A {

    default boolean equals(Object obj) {
        return true;
    }
}

class B implements A {

}
```

Let's suppose we have an interface, A, which declares a default method equals with the same signature as the equals method of the object class. Class B implements the A interface, which, by design, inherits the object class, and then, what version of the equals method does B inherit? A conflict would arise in this case.

Another reason is that, in this case, the interface default method becomes useless, because the equals method from the object class will always be invoked in children classes. This would conflict with the fact that the interface can be evolved

12. **What are all the advantages of Functional Interface?**

    - ✓ Lambda Expression
    - ✓ Default and static methods
    - ✓ Achieve the Functional Programming.

13. **How lambda expression and functional interfaces are related?**

    Lambda expressions can only be applied to abstract method of functional interface. With the help of lambda expression and functional interface only functional programming can be achieved.

14. **Can we create our own functional interface?**

    Yes, you can create your own functional interface. Java can implicitly identify functional interface but you can also annotate it with @FunctionalInterface

15. **Explain few Java 8 predefined Functional Interfaces?**

    **Predicate:** Predicate <T> interface is a functional interface with a method test (Object) to return a Boolean value. This interface signifies that an object is tested to be true or false.

```
List<Employee> employeeList = new ArrayList<>();

Employee employee = new Employee();
employee.setName("Sunny");
employee.setSalary(3000);
employeeList.add(employee);

Employee employee1 = new Employee();
employee1.setName("Bunny");
employee1.setSalary(2000);
employeeList.add(employee1);

Employee employee2 = new Employee();
employee2.setName("Munny");
employee2.setSalary(1000);
employeeList.add(employee2);

Predicate<Employee> employeePredicate = salary -> salary.getSalary() > 2000;

employeeList.forEach((emp)->{
    if(employeePredicate.test(emp)){
        System.out.println(emp.getName()); //Sunny
    }
});
```

This will be more useful in evaluating the chain of conditions like null checks.

Predicate below default Methods:

**and:** It will be useful in predicate chain if first predicate is false, then the other predicate is not evaluated.
**negate**: It returns a predicate that represents the logical negation of this predicate.
**or**: It will be use in predicate chain if first predicate is true, then the other predicate is not evaluated.
**isEqual:** It returns a predicate that tests if two arguments are equal according to Objects.equals(Object, Object).

**BiPredicate<T,U>:** It will take the two arguments and perform the validations and return the boolean value

T: denotes the type of the first input argument to the operation
U: denotes the type of the second input argument to the operation
R: denotes the Result

```
List<Employee> employeeList = new ArrayList<>();

Employee employee = new Employee();
employee.setName("Sunny");
employee.setSalary(3000);
employeeList.add(employee);

Employee employee1 = new Employee();
employee1.setName("Bunny");
employee1.setSalary(2000);
employeeList.add(employee1);

Employee employee2 = new Employee();
employee2.setName("Munny");
employee2.setSalary(1000);
employeeList.add(employee2);

BiPredicate<Employee, Employee> empBiPredicate = (salary,name)-> salary.getSalary() >2000 && !  name.getName().equals("Bunny");

employeeList.forEach(emp->{
    if(empBiPredicate.test(emp, emp)){
        System.out.println(emp.getName()); //Sunny
    }
});
```

**Function<T,U>:** It will take one input and returns the result. It has one abstract method apply(object).

```
List<Student> studentList = new ArrayList<>();

Student student = new Student();
student.setId(105);
student.setName("Test");
student.setAddress("Hyderabad");
student.setSalary(25000);

Student student1 = new Student();
student1.setId(102);
student1.setName("Sunny");
student1.setAddress("Tenali");
student1.setSalary(95000);

studentList.add(student);
studentList.add(student1);

Function<Student, Student> studentFunction = studentSal -> studentSal.getSalary() > 25000 ? studentSal : null;

studentList.forEach((studentData) -> {
    Student latestData = studentFunction.apply(studentData);
    System.out.println(latestData != null ? latestData.getName() : null);// null Sunny
});
```

## Default Methods of Function:

**andThen:** It returns a composed function that first applies this function to its input, and then applies the after function to the result. If evaluation of either function throws an exception, it is relayed to the caller of the composed function.

**compose:** It Returns a composed function that first applies the before function to its input, and then applies this function to the result. If evaluation of either function throws an exception, it is relayed to the caller of the composed function.

```
Function<Integer, Integer> multiply = (value) -> value * 2;
Function<Integer, Integer> add      = (value) -> value + 3;

Function<Integer, Integer> addThenMultiply = multiply.compose(add);

Integer result1 = addThenMultiply.apply(3);
System.out.println(result1);
```

**BiFunction<T,U,R>:** Its will take two arguments and returns the result. It has one abstract method apply(object,object).

```
class Employee {
    int id;
    String name;

    public Employee(int id, String name) {
        this.id = id;
        this.name = name;
    }
}

public class Java8BiFunction {
    public static void main(String[] args) {
        List<Employee> empList = new ArrayList<>();

        BiFunction<Integer, String, Employee> biFunctionEmployee = (id, name) -> new Employee(id, name);

        empList.add(biFunctionEmployee.apply(100, "Test"));
        empList.add(biFunctionEmployee.apply(200, "Abe"));
        empList.add(biFunctionEmployee.apply(300, "Ramu"));
        empList.add(biFunctionEmployee.apply(400, "Bdc"));

        empList.forEach((emp) -> {
            System.out.println(emp.id + " : " + emp.name); //100 : Test 200 : Abe 300 : Ramu 400 : Bdc
        });
    }
```

**Consumer<T>:** Consumer will take input but it does not return any value. It has one abstract method accept(object).

```
List<Student> studentList = new ArrayList<>();

Student student = new Student();
student.setId(105);
student.setName("Test");
student.setAddress("Hyderabad");
student.setSalary(25000);

Student student1 = new Student();
student1.setId(102);
student1.setName("Sunny");
student1.setAddress("Tenali");
student1.setSalary(95000);

studentList.add(student);
studentList.add(student1);

// Predicate
Predicate<Student> salPredicate = sal -> sal.getSalary() > 25000;

// Function
Function<Student, Student> studentFunction = studentSal -> salPredicate.test(studentSal) ? studentSal : null;

// Consumer
Consumer<Student> consumer = studt -> {
    System.out.println(studt.getName()); //null Sunny
};

studentList.forEach((studentData) -> {
    Student latestData = studentFunction.apply(studentData);
    consumer.accept(null != latestData ? latestData : new Student());
});
```

Default Methods:

**andThen:** It returns a composed Consumer that performs, in sequence, this operation followed by the after operation. If performing either operation throws an exception, it is relayed to the caller of the composed operation. If performing this operation throws an exception, the after operation will not be performed.

**BiConsumer<T,U> :** It will take two arguments and returns nothing.

```
class Department {
    int salary;
    String name;

    public Department(int salary, String name) {
        this.salary = salary;
        this.name = name;
    }
}

public class Java8BiConsumer {

    public static void main(String[] args) {
        List<Department> empList = new ArrayList<>();
        empList.add(new Department(1000,"Sunny"));
        empList.add(new Department(2000,"Bunny"));
        empList.add(new Department(3000,"Ramu"));
        BiConsumer<Department, Integer> biConsumer = (e,sal) -> e.salary = e.salary + 500;

        empList.forEach((empData)->{
            biConsumer.accept(empData, 500);
        });

        empList.forEach((emp)->{
            System.out.println(emp.name + " : " + emp.salary); //Sunny : 1500 Bunny : 2500 Ramu : 3500
        });
    }
```

**Supplier:** Supplier does not take any input but it will return the output. It has a single abstract method get

```java
public class Java8Supplier {
    public static void main(String[] args) {
        Supplier<Date> supplier = () -> new Date();
        System.out.println(supplier.get()); //Sat Apr 10 18:51:13 IST 2021
    }
}
```

16. **When to use the Consumer? Give a real time example?**

    Best way is printing something in loggers and loops iteration

17. **When to use Supplier? Give real time example?**

    If we are passing some input to some method with numbers as input if in case numbers was passed as null then we can use to supplier to give the input.

18. **Explain about LongPredicate, DoublePredicate?**

    **LongPredicate:** is a functional interface defined in java.util.function package. This interface can be used mainly for evaluating an input of type long and returns an output of type boolean.

    **DoublePredicate**: is a functional interface defined in java.util.function package. This interface can be used mainly for evaluating an input of type long and returns an output of type boolean.

19. **Explain about LongFunction, LongToDoubleFunction?**

    **LongFunction**: It represents a function which takes in a long-valued argument and produces a result of type R.

    **LongToDoubleFunction:** It represents a function which takes in a long-valued argument and gives a double-valued result.

20. **What is meant by method reference?**

    Java provides a new feature called method reference in Java 8. Method reference is used to refer method of functional interface. Method reference is a shorthand notation of a lambda expression to call a method.

    You use lambda expressions to create anonymous methods. Sometimes, however, a lambda expression does nothing but call an existing method. In those cases, it's often clearer to refer to the existing method by name. Method references enable you to do this; they are compact, easy-to-read lambda expressions for methods that already have a name.

    The method references can only be used to replace a single method of the lambda expression.

    Syntax:
    Object :: methodName

Example: With or without method reference

```
@FunctionalInterface
interface Mreference {
    void display();

}

public class MethodReferenceBasic {

    public void myMethod() {
        System.out.println("Instance Method");
    }

    public static void main(String[] args) {

        // Using Lambda Expressions
        Mreference mreference = () -> System.out.println("With Lambda Expressions");
        mreference.display(); // With Lambda Expressions

        // With Method Reference
        MethodReferenceBasic methodReferenceBasic = new MethodReferenceBasic();
        Mreference mreference2 = methodReferenceBasic::myMethod;
        mreference2.display();// Instance Method
```

Example: Method Reference For loop

```
public class MethodReferenceBasic {

    public static void main(String[] args) {
        List<Integer> intList = new ArrayList<>();
        intList.add(1);
        intList.add(2);
        intList.add(3);

        //Lambda Expressions
        intList.forEach(item -> System.out.println(item)); //1 2 3

        //Method Reference
        intList.forEach(System.out::println); //1 2 3
    }
```

**21. Explain the different types of Method References with example?**

We have three types of methods references,

- Instance Method Reference (object::instanceMethod)
- Instance method of an arbitrary object Method Reference (Class::instanceMethod)
- Static Method Reference (Class::staticMethod)
- Constructor Method Reference (Class::new)

Example: Instance Method Reference

```java
@FunctionalInterface
interface Say{
    void sayHello();
}

public class InstanceMethodReference {

    public void sayHi() {
        System.out.println("Invoking Instance Method");
    }

    public static void main(String[] args) {

        InstanceMethodReference instanceMethodReference = new InstanceMethodReference();

        Say say = instanceMethodReference::sayHi;
        say.sayHello(); //Invoking Instance Method
```

Example: Instance method of an arbitrary object Method Reference

```java
public class InstanceOrbitory {

    public static void main(String[] args) {

        List<String> personList = new ArrayList<>();

        personList.add("vicky");
        personList.add("poonam");
        personList.add("sachin");

        Collections.sort(personList,String::compareToIgnoreCase);

        personList.forEach(System.out::println); //poonam sachin vivky
    }
```

Example: Static Method Reference

```java
class Calculate{
    public static int multiply(int i, int j) {
        return i * j;
    }
}

public class StaticMethodRef {

    public static void main(String[] args) {

        BiFunction<Integer, Integer, Integer> biFunction = Calculate::multiply;
        System.out.println( biFunction.apply(5, 7));//35
    }
```

Example: Constructor Method Reference

```
@FunctionalInterface
interface Messageable {
    Message getMessage(String msg);
}

class Message {
    Message(String msg) {
        System.out.print(msg);
    }
}

public class ConstructorMethodRef {

    public static void main(String[] args) {

        Messageable messageable = Message::new;
        messageable.getMessage("Hello ......"); //Hello ......

    }

}
```

**22. What is the use of default methods in interface and give an example?**

Before Java 8, interfaces could have only abstract methods. The implementation of these methods has to be provided in a separate class. So, if a new method is to be added in an interface, then its implementation code has to be provided in the class implementing the same interface. To overcome this issue, Java 8 has introduced the concept of default methods which allow the interfaces to have methods with implementation without affecting the classes that implement the interface.

The default methods were introduced to provide backward compatibility so that existing interfaces can use the lambda expressions without implementing the methods in the implementation class. Default methods are also known as defender methods or virtual extension methods.

```
public interface TimeClient {
    void setTime(int hour, int minute, int second);
    void setDate(int day, int month, int year);
    void setDateAndTime(int day, int month, int year,
        int hour, int minute, int second);
    LocalDateTime getLocalDateTime();
    ZonedDateTime getZonedDateTime(String zoneString);
}
```

In the above example old Api using the different methods to get the Time, Date, DateAndTime, later new Date Api introduced to get the Date and Time in single shot then in future one more new api came to get the Zoned specific Date and time with single API. With the help of default methods we

can resolve the above problems without touching the old classes. This is called backward compatability.

```java
@FunctionalInterface
interface A {
    String sayHello(String name);

    default String defaultMethod(String name) {
        System.out.println("Default implemantation in base interface " + name); //Default implemantation in base interface Sunny
        return name;
    }
}

public class DefaultMethod implements A {

    public static void main(String[] args) {
        DefaultMethod defaultMethod = new DefaultMethod();
        System.out.println(defaultMethod.sayHello("Hello")); //Hello
        defaultMethod.defaultMethod("Sunny");
    }

    public String sayHello(String name) {
        return name;
    }

}
```

**23. How to resolve the Multiple Inheritance problem with default methods with an example?**

When a class implements several interfaces that define the same default methods code will not compile and will get the ambiguity problems.

To solve this ambiguity, we must explicitly provide an implementation for the methods:

```java
interface TestInterface1 {
    default void show() {
        System.out.println("Default TestInterface1");
    }
}

interface TestInterface2 {
    default void show() {
        System.out.println("Default TestInterface2");
    }
}

public class DefaultMethod implements TestInterface1, TestInterface2 {

    public static void main(String[] args) {
        DefaultMethod defaultMethod = new DefaultMethod();
        defaultMethod.show();
    }

    public void show() {
        TestInterface1.super.show(); //Default TestInterface1
        TestInterface2.super.show(); //Default TestInterface2
    }
```

**24. Difference between abstract classes and Java 8 interfaces?**

| | Interface | Abstract Class |
|---|---|---|
| Constructors | ✗ | ✓ |
| Static Fields | ✓ | ✓ |
| Non-static Fields | ✗ | ✓ |
| Final Fields | ✓ | ✓ |
| Non-final Fields | ✗ | ✓ |
| Private Fields & Methods | ✗ | ✓ |
| Protected Fields & Methods | ✗ | ✓ |
| Public Fields & Methods | ✓ | ✓ |
| Abstract methods | ✓ | ✓ |
| Static Methods | ✓ | ✓ |
| Final Methods | ✗ | ✓ |
| Non-final Methods | ✓ | ✓ |
| Default Methods | ✓ | ✗ |

**25. Explain Static methods in interface?**

Static Methods in Interface are those methods, which are defined in the interface with the keyword static. Unlike other methods in Interface, these static methods contain the complete definition of the function and since the definition is complete and the method is static, therefore these methods cannot be overridden or changed in the implementation class.

Similar to Default Method in Interface, the static method in an interface can be defined in the interface, but cannot be overridden in Implementation Classes. To use a static method, Interface name should be instantiated with it, as it is a part of the Interface only.

```
interface Calculate{
    public static int multiply(int i, int j) {
        return i * j;
    }
}

public class StaticMethodRef implements Calculate {

    public static void main(String[] args) {

        BiFunction<Integer, Integer, Integer> biFunction = Calculate::multiply;
        System.out.println( biFunction.apply(5, 7));//35
    }
```

**26. What is the need of Optional?**

Every Java Programmer is familiar with NullPointerException. It can crash your code. And it is very hard to avoid it without using too many null checks.

Java 8 has introduced a new class Optional in java.util package. It can help in writing a neat code without using too many null checks. By using Optional, we can specify alternate values to return or alternate code to run. This makes the code more readable because the facts which were hidden are now visible to the developer.

**27. Explain few methods of Optional?**

**empty():** It returns an empty Optional object. No value is present for this Optional.
**of(T value) :** It returns an Optional with the specified present non-null value. Otherwise it will throw the Null pointer Exception.
**ofNullable(T value):** It returns an Optional describing the specified value, if non-null, otherwise returns an empty Optional.
**get():** If a value is present in this Optional, returns the value, otherwise throws NoSuchElementException.
**isPresent():** It returns true if there is a value present, otherwise false.
**orElse(T other):** It returns the value if present, otherwise returns other.
**orElseThrow():** It returns the contained value, if present, otherwise throw an exception to be created by the provided supplier.

**28. Given an example for Optional methods?**

```
public static void main(String[] args) {
    String[] str = new String[10];
    str[5] = "JAVA OPTIONAL CLASS EXAMPLE";

    Optional<String> empty = Optional.empty();
    System.out.println(empty); // Optional.empty

    Optional<String> value = Optional.of(str[5]);
    Optional<String> value1 = Optional.ofNullable(str[5]);
    System.out.println("With string value: " + value.filter((s) -> s.equals("And")));// With string value:
                                                                                    // Optional.empty
    System.out.println("With String value: " + value1.filter((s) -> s.equals("And")));// With string value:
                                                                                    // Optional.empty

    str[5] = null;
    // Optional<String> withNull = Optional.of(str[5]);
    Optional<String> withNull1 = Optional.ofNullable(str[5]);

    // System.out.println("With string value: " + withNull.filter((s) ->
    // s.equals("Java")));//java.lang.NullPointerException
    System.out.println("With Null Value: " + withNull1.filter((s) -> s.equals("Java"))); // With Null Value:
                                                                                    // Optional.empty

    System.out.println("Getting value: " + value.get()); // Getting value: JAVA OPTIONAL CLASS EXAMPLE
    System.out.println("Is value present: " + value.isPresent()); // Is value present: true
    System.out.println("orElse: " + value.orElse("Value is not present")); // orElse: JAVA OPTIONAL CLASS EXAMPLE
}
```

**29. What is the Use of Optional.empty()?**

If you want to return the empty object then will use the Optional.empty().

**30. When will go for Optional.of()?**

If you are sure about the value presented in output object then will go for Optional.of(), Otherwise it will throw the NullPointerException.

**31. When will go for Optional.ofNullable()?**

If you are sure or not sure about the value presented in output object then will go for Optional.ofNullable(). If value presented it will return the value otherwise it will return the empty Optional.

**32. What is the use of forEach() given an example?**

The Java forEach() method is a utility function to iterate over a collection such as (list, set or map) and stream.

```
public static void main(String[] args) {

    List<Integer> intList = new ArrayList<>();
    intList.add(1);
    intList.add(2);
    intList.add(3);

    //Iterating the List
    intList.forEach(System.out::println); //1 2 3

    //Iterating over the stream
    intList.stream().forEach(num -> System.out.println(num)); //123

    Set<Integer> intSet = new HashSet<>();
    intSet.add(1);
    intSet.add(2);
    intSet.add(3);

    //Iterating the Set
    intSet.forEach(System.out::println); //123


    Map<Integer, String> map = new HashMap<>();
    map.put(101, "Sunny");
    map.put(102, "Bunny");

    //Iterating Map
    map.forEach((key,value) ->{
        System.out.println("Key " + key); //101 102
        System.out.println("Value " + value); //Sunny Bunny
    });
```

**33. What is the use of StringJoiner() give an example?**

Java added a new final class StringJoiner in java.util package. It is used to construct a sequence of characters separated by a delimiter. Now, you can create string by passing delimiters like comma(,), hyphen(-) etc. You can also pass prefix and suffix to the char sequence.

**Constructors:**

**----------**

**StringJoiner(CharSequence delimiter):** It constructs a StringJoiner with no characters in it, with no prefix or suffix, and a copy of the supplied delimiter. It throws NullPointerException if delimiter is null.

**StringJoiner(CharSequence delimiter,CharSequence prefix,CharSequence suffix):** It constructs a StringJoiner with no characters in it using copies of the supplied prefix, delimiter and suffix. It throws NullPointerException if prefix, delimiter, or suffix is null.

```
public static void main(String[] args) {

    List<String> intList = new ArrayList<>();
    intList.add("1");
    intList.add("2");
    intList.add("3");

    StringJoiner commaJoiner = new StringJoiner(",");

    intList.forEach(num -> commaJoiner.add(num));
    System.out.println(commaJoiner);//1,2,3

    StringJoiner joiner = new StringJoiner(",","-","$");
    intList.forEach(num -> joiner.add(num));
    System.out.println(joiner);//-1,2,3$
```

**34. Explain the methods of String Joiner?**

**add:** It adds a copy of the given CharSequence value as the next element of the StringJoiner value. If newElement is null,"null" is added.
**merge:** It adds the contents of the given StringJoiner without prefix and suffix as the next element if it is non-empty. If the given StringJoiner is empty, the call has no effect.
**length:** It returns the length of the String representation of this StringJoiner.
**setEmptyValue:** It sets the sequence of characters to be used when determining the string representation of this StringJoiner and no elements have been added yet, that is, when it is empty.

```
List<String> intList = new ArrayList<>();
intList.add("1");
intList.add("2");
intList.add("3");

StringJoiner commaJoiner = new StringJoiner(",");

intList.forEach(num -> commaJoiner.add(num));
System.out.println(commaJoiner);// 1,2,3

StringJoiner joiner = new StringJoiner(",", "-", "$");
intList.forEach(num -> joiner.add(num));
System.out.println(joiner);// -1,2,3$

System.out.println("Merge Method " + commaJoiner.merge(joiner));// Merge Method 1,2,3,1,2,3
System.out.println("Length Method " + commaJoiner.length()); // Length Method 11

StringJoiner emptyJoiner = new StringJoiner(",");

System.out.println("Setting empty value " + emptyJoiner.setEmptyValue("Setting Empty Value"));//Setting empty value Setting Empty Value
```

**35. What is meant by stream?**

Stream is a sequence of elements from source.

Stream does not store elements. Operations performed on a stream does not modify its source.

For example, filtering a Stream obtained from a collection produces a new Stream without the filtered elements, rather than removing elements from the source collection.

**36. Explain the features of Stream?**

- ✓ Stream does not store elements.
- ✓ Stream is functional in nature. Operations performed on a stream does not modify it's source. For example, filtering a Stream obtained from a collection produces a new Stream without the filtered elements, rather than removing elements from the source collection.
- ✓ The elements of a stream are only visited once during the life of a stream. Like an Iterator, a new stream must be generated to revisit the same elements of the source.
- ✓ Stream is lazy and evaluates code only when required.

**37. Difference between Java Stream and Collection?**

All of us have watched online videos on Youtube. When we start watching a video, a small portion of the file is first loaded into the computer and starts playing. We don't need to download the complete video before we start playing it. This is called streaming.

At a very high level, we can think of that small portions of the video file as a stream, and the whole video as a Collection.

At the granular level, the difference between a Collection and a Stream is to do with when the things are computed. A Collection is an in-memory data structure, which holds all the values that the data structure currently has. Every element in the Collection has to be computed before it can be added to the Collection. While a Stream is a conceptually a pipeline, in which elements are computed on demand.

Stream operations can either be executed sequentially or parallel. When performed parallelly, it is called a parallel stream.

**38. How to create the Streams?**

Streams can be created with the help of Stream.of(type) method.

**39. Give an example to build Stream of Integer, String?**

```
Stream<Integer> intStream = Stream.of(1,2,3,4,5);
intStream.forEach(System.out::println); //1 2 3 4 5

Stream<String> stringStream = Stream.of("A","B","C");
stringStream.forEach(System.out::println); //A B C
```

**40. Give an example to convert List, Array, Set and Map to Stream?**

```
int [] intArr = new int[] {1,2,3,4};

//Array to Stream
IntStream arayStream = Arrays.stream(intArr);
arayStream.forEach(System.out::println); //1 2 3 4

List<String> list = new ArrayList<>();
list.add("Abc");
list.add("Bcd");

//List to Stream
Stream<String> listToStream = list.stream();
listToStream.forEach(System.out::println); //Abc Bcd

Set<String> set = new HashSet<>();
set.add("Abc");
set.add("Bcd");

//Set to Stream
Stream<String> setToStream = set.stream();
setToStream.forEach(System.out::println); //Abc Bcd

Map<Integer, String> map = new HashMap<>();
map.put(101, "Sunny");
map.put(102, "Bunny");

//Map to Stream
Stream<Map.Entry<Integer, String>> mapToStream = map.entrySet().stream();

//Iterating Map
mapToStream.forEach(data ->{
    System.out.println(data.getKey()); //101 102
    System.out.println(data.getValue()); //Sunny Bunny
});
```

**41. Give an example to collect Stream elements to a List and Unmodifiable List?**

```
List<String> list = new ArrayList<>();
list.add("A");
list.add("B");
list.add("A");
list.add("C");

//Collecting Normal List
List<String> distinctList = list.stream().distinct().collect(Collectors.toList());

System.out.println(distinctList); //[A, B, C]

//Collecting Unmodifiable List
distinctList = list.stream().distinct().collect(Collectors.toUnmodifiableList());

System.out.println(distinctList); //[A, B, C]
```

**42. Give an example to collect Stream elements to a Set and Unmodifiable Set?**

```
Set<String> set = new HashSet<>();
set.add("A");
set.add("B");
set.add("A");
set.add("C");

//Collecting Normal Set
Set<String> distinctSet = set.stream().distinct().collect(Collectors.toSet());

System.out.println(distinctSet); //[A, B, C]

//Collecting Unmodifiable List
distinctSet = set.stream().distinct().collect(Collectors.toUnmodifiableSet());

System.out.println(distinctSet); //[C, B, A]
```

**43. Give an example to collect Stream elements to a Map?**

```
public static void main(String[] args) {
    Map<Integer, String> map = new HashMap<>();
    map.put(101, "Sunny");
    map.put(102, "Bunny");

    //Map to Stream
    Stream<Map.Entry<Integer, String>> mapToStream = map.entrySet().stream();

    //Stream to Map
    Map<Integer,String> convertedMap = mapToStream.collect(Collectors.toMap(Map.Entry::getKey, Map.Entry::getValue));

    convertedMap.forEach((key,value)->{
        System.out.println(key); //101 102
        System.out.println(value); //Sunny Bunny
    });
```

**44. Give an example to filter the data of Stream?**

The 'filter' method is used to eliminate elements based on a criteria.

Example: If we have list of employee's data and you want to get the employees details with salary > 5000 then we need to filter with the salary parameter

```
List<Employee> employeeList = new ArrayList<>();

Employee employee = new Employee();
employee.setName("Sunny");
employee.setSalary(2000);
employeeList.add(employee);

Employee employee1 = new Employee();
employee1.setName("Bunny");
employee1.setSalary(1000);
employeeList.add(employee1);

Employee employee2 = new Employee();
employee2.setName("Munny");
employee2.setSalary(5000);
employeeList.add(employee2);

//Filtering Employee salary > 1000
List<Employee> filteredList = employeeList.stream().filter(emp -> emp.getSalary()>1000).collect(Collectors.toList());

filteredList.forEach(empData -> System.out.println(empData.getName())); //Sunny Munny
```

**45. Give an example to limit the data of Stream?**

The 'limit' method is used to reduce the size of the stream.

Example: If we have 10 records in list if we want to get only two records we need to pass the number 2 as argument to limit method

```
List<Employee> employeeList = new ArrayList<>();

Employee employee = new Employee();
employee.setName("Sunny");
employee.setSalary(2000);
employeeList.add(employee);

Employee employee1 = new Employee();
employee1.setName("Bunny");
employee1.setSalary(1000);
employeeList.add(employee1);

Employee employee2 = new Employee();
employee2.setName("Munny");
employee2.setSalary(5000);
employeeList.add(employee2);

//Getting the first two employees data only
List<Employee> filteredList = employeeList.stream().limit(2).collect(Collectors.toList());

filteredList.forEach(empData -> System.out.println(empData.getName())); //Sunny Munny
```

**46. Given an example to get the unique the data of Stream?**

**distinct:** It does not take any argument and returns the distinct elements in the stream, eliminating duplicates. It uses the equals() method of the elements to decide whether two elements are equal or not

```java
List<String> list = new ArrayList<>();
list.add("A");
list.add("B");
list.add("A");
list.add("C");

//Collecting Normal List
List<String> distinctList = list.stream().distinct().collect(Collectors.toList());

System.out.println(distinctList); //[A, B, C]
```

**47. What is the use of Stream.map() and give an example?**

map: produces a new stream after applying a function to each element of the original stream. The new stream could be of different type.

Stream map(Function mapper) is an intermediate operation. Intermediate operations are invoked on a Stream instance and after they finish their processing, they give a Stream instance as output.

**Example:** In the list of employees if you want to retunes their names only with Upper case or if we want to return the sum of salary of all employees then will go for map function of stream. Map function will return different stream like above String of names or sum of salary Int.

```java
List<Employee> employeeList = new ArrayList<>();

Employee employee = new Employee();
employee.setName("Sunny");
employee.setSalary(2000);
employeeList.add(employee);

Employee employee1 = new Employee();
employee1.setName("Bunny");
employee1.setSalary(1000);
employeeList.add(employee1);

Employee employee2 = new Employee();
employee2.setName("Munny");
employee2.setSalary(5000);
employeeList.add(employee2);

//Converting names to to upper case
List<String> filteredList = employeeList.stream().map(emp -> emp.getName().toUpperCase()).collect(Collectors.toList());
filteredList.forEach(empData -> System.out.println(empData)); //SUNNY BUNNY MUNNY

int salarySum = employeeList.stream().mapToInt(emp -> emp.getSalary()).sum();
System.out.println(salarySum); //8000
```

**48. What is the use of Stream.sorted()and give an example?**

**sorted:** The sorted() method is an intermediate operation that returns a sorted view of the stream. The elements in the stream are sorted in natural order unless we pass a custom Comparator.

```java
public static void main(String[] args) {
    List<String> stringList = new ArrayList<>();
    stringList.add("Abc");
    stringList.add("Ade");
    stringList.add("Acd");

    List<String> sortedList = stringList.stream().sorted().collect(Collectors.toList());

    sortedList.forEach(System.out::println); //Abc Acd Ade
}
```

**49. What is the use of Stream.allMatch() and give an example?**

**allMatch:** It returns all elements of this stream which match the provided predicate. It will return the boolean value as result.

**If the stream is empty then true is returned and the predicate is not evaluated.**

**Example:** As per allMatch the condition provided should be match with al records. If all records matched will get true otherwise false. As per below example one of the employee Bunny salary is less than 1000 that why we are seeing the result as false.

```java
List<Employee> employeeList = new ArrayList<>();

Employee employee = new Employee();
employee.setName("Sunny");
employee.setSalary(2000);
employeeList.add(employee);

Employee employee1 = new Employee();
employee1.setName("Bunny");
employee1.setSalary(1000);
employeeList.add(employee1);

Employee employee2 = new Employee();
employee2.setName("Munny");
employee2.setSalary(5000);
employeeList.add(employee2);

//Checking Salary match
boolean matchFound = employeeList.stream().allMatch(emp -> emp.getSalary()>1000);

System.out.println(matchFound); //false
```

**50. What is the use of Stream.anyMatch() and give an example?**

**anyMatch:** It will return true if any of the value is matched in records.

**If the stream is empty then false is returned and the predicate is not evaluated.**

**Example:** As per below example two employees salary is greater than 1000 that why we are seeing the result as true

```
List<Employee> employeeList = new ArrayList<>();

Employee employee = new Employee();
employee.setName("Sunny");
employee.setSalary(2000);
employeeList.add(employee);

Employee employee1 = new Employee();
employee1.setName("Bunny");
employee1.setSalary(1000);
employeeList.add(employee1);

Employee employee2 = new Employee();
employee2.setName("Munny");
employee2.setSalary(5000);
employeeList.add(employee2);

//Checking Salary match
boolean matchFound = employeeList.stream().anyMatch(emp -> emp.getSalary()>1000);

System.out.println(matchFound); //true
```

**51. What is the use of Stream.noneMatch() and give an example?**

noneMatch: It will return the true value if the mentioned criteria element is not matching in the records. If any of the record is matched as per criteria it will return true.
**If the stream is empty then true is returned and the predicate is not evaluated.**

**Example:** As per below example all the employees salary is less than 10000 that why we are seeing the result as true

```
List<Employee> employeeList = new ArrayList<>();

Employee employee = new Employee();
employee.setName("Sunny");
employee.setSalary(2000);
employeeList.add(employee);

Employee employee1 = new Employee();
employee1.setName("Bunny");
employee1.setSalary(1000);
employeeList.add(employee1);

Employee employee2 = new Employee();
employee2.setName("Munny");
employee2.setSalary(5000);
employeeList.add(employee2);

//Checking Salary match
boolean matchFound = employeeList.stream().noneMatch(emp -> emp.getSalary()>10000);

System.out.println(matchFound); //true
```

**52. What is the use of Stream.reduce() and give an example?**

A reduction operation (also called as fold) takes a sequence of input elements and combines them into a single summary result by repeated application of a combining operation.

**Syntax:**
T reduce(T identity, BinaryOperator<T> accumulator);

identity = default or initial value.
BinaryOperator = functional interface, take two values and produces a new value.

```
int[] numbers = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

System.out.println(Arrays.stream(numbers).reduce(0, (a, b) -> a + b));// 55

// Shortcut
System.out.println(Arrays.stream(numbers).reduce(0, Integer::sum)); // 55

List<Invoice> invoicesList = new ArrayList<>();

Invoice invoice = new Invoice();
invoice.setQuantity(BigDecimal.valueOf(3));
invoice.setAmount(BigDecimal.valueOf(200));
invoicesList.add(invoice);

Invoice invoice1 = new Invoice();
invoice1.setQuantity(BigDecimal.valueOf(2));
invoice1.setAmount(BigDecimal.valueOf(300));
invoicesList.add(invoice1);

BigDecimal sum = invoicesList.stream().map(item -> item.getAmount().multiply(item.getQuantity())).reduce(BigDecimal.ZERO,BigDecimal::add);
System.out.println(sum);//1200
```

**53. What is the use of Stream.findFirst() and give an example?**

**findFirst()** returns an Optional (a container object which may or may not contain a non-null value) describing the first element of this stream, or an empty Optional if the stream is empty.

```
List<Employee> employeeList = new ArrayList<>();

Employee employee = new Employee();
employee.setName("Sunny");
employee.setSalary(2000);
employeeList.add(employee);

Employee employee1 = new Employee();
employee1.setName("Bunny");
employee1.setSalary(1000);
employeeList.add(employee1);

Employee employee2 = new Employee();
employee2.setName("Munny");
employee2.setSalary(5000);
employeeList.add(employee2);

//Checking Salary match

Optional<Employee> firstElement =  employeeList.stream().findFirst();

if(firstElement.isPresent()) {
    System.out.println(firstElement.get().getName()); //Sunny
}
```

In the above example if we pass the empty list and did not put the isPresent condition then will get java.util.NoSuchElementException: No value present.

**54. What is the use of Stream.findAny() give an example?**

It returns an Optional describing some element of the stream, or an empty Optional if the stream is empty.

findAny() is a terminal-short-circuiting operation of Stream interface. This method returns first element satisfying the intermediate operations.

```
List<Employee> employeeList = new ArrayList<>();

Employee employee = new Employee();
employee.setName("Sunny");
employee.setSalary(2000);
employeeList.add(employee);

Employee employee1 = new Employee();
employee1.setName("Bunny");
employee1.setSalary(1000);
employeeList.add(employee1);

Employee employee2 = new Employee();
employee2.setName("Munny");
employee2.setSalary(5000);
employeeList.add(employee2);

//Checking Salary match

Optional<Employee> firstElement =  employeeList.stream().findAny();

if(firstElement.isPresent()) {
    System.out.println(firstElement.get().getName()); //Sunny
}
```

In the above example if we pass the empty list and did not put the isPresent condition then will get java.util.NoSuchElementException: No value present.

**55. What is the use of parallelStream give an example?**

Normally any java code has one stream of processing, where it is executed sequentially. Whereas by using parallel streams, we can divide the code into multiple streams that are executed in parallel on separate cores and the final result is the combination of the individual outcomes. The order of execution, however, is not under our control.

Therefore, it is advisable to use parallel streams in cases where no matter what is the order of execution, the result is unaffected and the state of one element does not affect the other as well as the source of the data also remains unaffected.

```
public static void main(String[] args) {
    System.out.println("Normal...");

    IntStream range = IntStream.rangeClosed(1, 2);
    System.out.println(range.isParallel()); //true
    range.forEach(x -> System.out.println("Thread : " + Thread.currentThread().getName() + ", value: " + x));
    //Thread : main, value: 1 Thread : main, value: 2

    System.out.println("Parallel...");

    IntStream range2 = IntStream.rangeClosed(1, 2);
    IntStream range2Parallel = range2.parallel();
    System.out.println(range2Parallel.isParallel());  //true
    range2.parallel()
            .forEach(x -> System.out.println("Thread : " + Thread.currentThread().getName() + ", value: " + x));
    //Thread : main, value: 2 Thread : ForkJoinPool.commonPool-worker-3, value: 1
}
```

**56. What is the use of flatMap() and give an example?**

A stream can hold complex data structures like Stream<List<String>>. In cases like this, flatMap() helps us to flatten the data structure to simplify further operations

```
 // Creating a list of Prime Numbers
List<Integer> primeNumbers = Arrays.asList(5, 7, 11,13);

// Creating a list of Odd Numbers
List<Integer> oddNumbers = Arrays.asList(1, 3, 5);

// Creating a list of Even Numbers
List<Integer> evenNumbers = Arrays.asList(2, 4, 6, 8);

List<List<Integer>> listOfListofInts =
        Arrays.asList(primeNumbers, oddNumbers, evenNumbers);

System.out.println("The Structure before flattening is : " +
        listOfListofInts); //The Structure before flattening is : [[5, 7, 11, 13], [1, 3, 5], [2, 4, 6, 8]]

//Using Flat Map
List<Integer> afterFaltMap = listOfListofInts.stream().flatMap(num -> num.stream()).collect(Collectors.toList());
System.out.println("After Flattern "+ afterFaltMap); //After Flattern [5, 7, 11, 13, 1, 3, 5, 2, 4, 6, 8]
```

**57. Difference between map and faltMap?**

| **map()** | **flatMap()** |
|---|---|
| Its intermediate stream operation and return another stream as method output | Its intermediate stream operation and return another stream as method output |
| map() function produces one output for one input value | flatMap() returns zero or more than zero values as output. |
| Only perform the mapping. | Perform mapping as well as flattening. |
| map() is used only for transformation. | flatMap() is used both for transformation and mapping. |

**58. What is the use of Stream.peek() and give an example?**

Returns a stream consisting of the elements of this stream, additionally performing the provided action on each element as elements are consumed from the resulting stream.

It is an intermediate operation. Using peek without any terminal operation does nothing.

This method exists mainly to support debugging, where you want to see the elements as they flow past a certain point in a pipeline.

**Example:** In the below example add the first 3 numbers to the list when we performed the peek operation I was returned current list data as well as terminal operation(Collected to new list) data also.

```
List<Integer> intList = new ArrayList<>();
intList.add(1);
intList.add(2);
intList.add(3);

List<Integer> listData = intList.stream().peek(System.out::println).collect(Collectors.toList());
System.out.println(listData);//1 2 3 [1, 2, 3]
```

**59. Explain about forEachOrdered() and give an example?**

Java Stream forEachOrdered(action) method is used to iterate over all the elements of the given Stream and to perform an Consumer action on the each element of the Stream, in the encounter order of the Stream if the Stream has a defined encounter order.

```
List<Integer> list = Arrays.asList(2, 4, 6, 8, 10);

list.stream()
    .sorted(Comparator.reverseOrder())
    .forEachOrdered(System.out::println); //10 8 6 4 2
```

**60. Difference between forEach() and forEachOrdered()?**

| **forEach()** | **forEachOrdered()** |
|---|---|
| It is used to iterate over the collections | It is used to iterate over the collections |
| For parallel streams, forEach() operation does not guarantee to respect the encounter order of the Stream | While the forEachOrdered() operation respects the encounter order of the Stream if the stream has a defined encounter order. This behavior is true for parallel streams as well as sequential streams. |

```
List<Integer> list = Arrays.asList(2, 4, 6, 8, 10);

list.stream().parallel().forEachOrdered(System.out::println); // 2 4 6 8 10

list.stream().parallel().forEach(System.out::println); //6 10 8 2 4
```

**61. Write a logic to sort the objects using Streams?**

Sorting on the Objects level can be happening by applying the sort() method on the list. Custom sorting can be applied by using the comparing method of Comparator interface. For comparing method we need to pass the field names of the object by using Method reference feature.

```
List<Employee> employeeList = new ArrayList<>();

Employee employee = new Employee();
employee.setName("Sunny");
employee.setSalary(2000);
employeeList.add(employee);

Employee employee1 = new Employee();
employee1.setName("Bunny");
employee1.setSalary(1000);
employeeList.add(employee1);

Employee employee2 = new Employee();
employee2.setName("Munny");
employee2.setSalary(5000);
employeeList.add(employee2);

//Added comparision logic on single fields
employeeList.sort(Comparator.comparing(Employee::getName));

employeeList.forEach(emp -> System.out.println(emp.getName() + " " + emp.getSalary())); // Bunny 1000 Munny 5000
                                                                                        // Sunny 2000
//Added comparision logic on multiple fields
employeeList.sort(Comparator.comparing(Employee::getSalary).thenComparing(Employee::getName));
employeeList.forEach(emp -> System.out.println(emp.getName() + " " + emp.getSalary())); // Bunny 1000 Sunny 2000
                                                                                        // Munny 5000
```

**62. What is the use of Infinite Stream?**

Streams are different from collections although they can be created from collections. Unlike collections, a stream can go on generating/producing values forever. Java 8 Streams API provides two static methods in the Stream interface for creating infinite streams. These are Stream.iterate() and Stream.generate().

Since infinite streams need to be limited to a finite number, based on specific requirement, hence it is a common practice to limit the number of elements produced by a stream using the Stream.limit() method.

**63. Write a logic to generate the Random Numbers using Streams?**

```
List<Integer> randomNumbers = Stream.generate(() -> (new Random()).nextInt(100))
        .limit(10)
        .collect(Collectors.toList());
System.out.println(randomNumbers); //[89, 44, 95, 5, 85, 80, 15, 32, 25, 10]
```

### 64. Write a logic to get the last element in Stream?

```
List<String> list = Arrays.asList("node", "java", "c++", "react", "javascript");

String result = list.stream().reduce((first, second) -> second).orElse("no last element");

System.out.println(result); //javascript
```

### 65. Write a logic to convert Iterator to Stream?

```
List<String> list = Arrays.asList("node", "java", "c++", "react", "javascript");
Stream<String> stream = StreamSupport.stream(list.spliterator(), false);
System.out.println(stream.collect(Collectors.toList()));//[node, java, c++, react, javascript]
```

### 66. Write a logic to sort map by value in descending order?

```
Map<String, Integer> unsortMap = new HashMap<>();
unsortMap.put("alex", 1);
unsortMap.put("david", 2);
unsortMap.put("elle", 3);
unsortMap.put("charles", 4);
unsortMap.put("brian", 5);

//LinkedHashMap preserve the ordering of elements in which they are inserted
LinkedHashMap<String, Integer> sortedMapByValue = new LinkedHashMap<>();

//Sorting Map By Value
unsortMap.entrySet().stream().sorted(Map.Entry.comparingByValue())
        .forEachOrdered(x -> sortedMapByValue.put(x.getKey(), x.getValue()));

System.out.println(sortedMapByValue); //{alex=1, david=2, elle=3, charles=4, brian=5}

LinkedHashMap<String, Integer> sortedMapByKey = new LinkedHashMap<>();

unsortMap.entrySet().stream().sorted(Map.Entry.comparingByKey()).forEach(data -> sortedMapByKey.put(data.getKey(), data.getValue()));

System.out.println(sortedMapByKey); //{alex=1, brian=5, charles=4, david=2, elle=3}
```

### 67. Give an example to sort map with custom comparator?

```
public static void main(String[] args) {
    Map<Integer, Employee> unsortMap = new HashMap<>();

    Employee employee = new Employee();
    employee.setName("Sunny");
    employee.setSalary(2000);
    employee.setEmpId(102);
    unsortMap.put(employee.getEmpId(), employee);

    Employee employee1 = new Employee();
    employee1.setName("Bunny");
    employee1.setSalary(1000);
    employee1.setEmpId(100);
    unsortMap.put(employee1.getEmpId(), employee1);

    Employee employee2 = new Employee();
    employee2.setName("Munny");
    employee2.setSalary(5000);
    employee2.setEmpId(101);
    unsortMap.put(employee2.getEmpId(), employee2);

    Comparator<Employee> byName = (Employee o1, Employee o2)-> o1.getName().compareTo(o2.getName());

    LinkedHashMap<Integer, Employee> sortedMapByKey = new LinkedHashMap<>();

    unsortMap.entrySet().stream().sorted(Map.Entry.<Integer, Employee>comparingByValue(byName)).forEachOrdered(data -> sortedMapByKey.put(data.getKey(), data.getValue()));

    sortedMapByKey.forEach((k,v)->{
        System.out.println(v.getEmpId() + " " + v.getName() + " " + v.getSalary());//100 Bunny 1000 101 Munny 5000 102 Sunny 2000
    });
}
```

**68. Write a logic to convert Array to list and list to array?**

```
List<String> stringList = new ArrayList<>();
stringList.add("Java");
stringList.add("C++");
stringList.add("JavaScript");

String [] strArr = stringList.stream().toArray(String[]::new);
System.out.println(strArr);

String [] strData = new String[] {"A","B","C"};
List<String> data = Arrays.asList(strData);
System.out.println(data); //[A, B, C]
```

**69. Explain the few methods of intermediate, terminal operations?**

**Intermediant Operations**

**----------**

**map:** produces a new stream after applying a function to each element of the original stream. The new stream could be of different type.
**filter:** The 'filter' method is used to eliminate elements based on a criteria.
**limit:** The 'limit' method is used to reduce the size of the stream.
**sorted:** The sorted() method is an intermediate operation that returns a sorted view of the stream. The elements in the stream are sorted in natural order unless we pass a custom Comparator.
**flatMap:** A stream can hold complex data structures like Stream<List<String>>. In cases like this, flatMap() helps us to flatten the data structure to simplify further operations
**peek:** peek() method returns a new Stream consisting of all the elements from the original Stream after applying a given Consumer action.Java program to use peek() API to debug the Stream operations and logging Stream elements as they are processed.
**distinct:** It does not take any argument and returns the distinct elements in the stream, eliminating duplicates. It uses the equals() method of the elements to decide whether two elements are equal or not

**Terminal Operations:**

**----------**

**forEach():** The forEach() method helps in iterating over all elements of a stream and perform some operation on each of them. The operation to be performed is passed as the lambda expression.
**collect:** method is used to receive elements from a steam and store them in a collection.
**match():** Various matching operations can be used to check whether a given predicate matches the stream elements. All of these matching operations are terminal and return a boolean result.
**count():** The count() is a terminal operation returning the number of elements in the stream as a long value.
**reduce():** A reduction operation (also called as fold) takes a sequence of input elements and combines them into a single summary result by repeated application of a combining operation.

**toArray:** We saw how we used collect() to get data out of the stream. If we need to get an array out of the stream, we can simply use toArray()
**allMatch:** It returns all elements of this stream which match the provided predicate. If the stream is empty then true is returned and the predicate is not evaluated.
**anyMatch:** It returns any element of this stream that matches the provided predicate. If the stream is empty then false is returned and the predicate is not evaluated.
**findFirst:** findFirst() returns an Optional for the first entry in the stream; the Optional can, of course, be empty
**findAny:** It returns an Optional describing some element of the stream, or an empty Optional if the stream is empty.

**70. Difference between intermediate and terminal operations?**

Intermediate operations are those operations that return Stream itself, allowing for further operations on a stream.
These operations are always lazy, i.e. they do not process the stream at the call site. An intermediate operation can only process data when there is a terminal operation. Some of the intermediate operations are filter, map and flatMap.
In contrast, terminal operations terminate the pipeline and initiate stream processing. The stream is passed through all intermediate operations during terminal operation call. Terminal operations include forEach, reduce, Collect and sum.

**71. Explain Lazy evaluation of Streams?**

In lazy processing, the operations are performed after the action. It is processing the data only on demand.
It is an important characteristic of streams because the operation on the source data is only performed when the terminal operation is initiated. It doesn't consume the source elements as in eager loading, the source elements are consumed only on demand.

As you have seen in Stream operations:

- ✓ The intermediate stream doesn't produce the result. They create a new Stream only.
- ✓ The terminal operation produces the result.

**72. What is the difference between enhanced forEach and Java 8 forEach?**

The main difference between them is that they are different iterators. The enhanced for-loop is an external iterator, whereas the new forEach method is internal.

**Internal Iterator — forEach:** This type of iterator manages the iteration in the background and leaves the programmer to just code what is meant to be done with the elements of the collection. The iterator instead manages the iteration and makes sure to process the elements one-by-one.

```
names.forEach(name -> System.out.println(name));
```

In the *forEach* method above, we can see that the argument provided is a lambda expression. This means that the method only needs to know **what is to be done**, and all the work of iterating will be taken care of internally.

**External Iterator — for-loop:** External iterators mix what and how the loop is to be done. Enumerations, Iterators and enhanced for-loop are all external iterators (remember the methods iterator(), next() or hasNext()?). In all these iterators, it's our job to specify how to perform iterations.

```
for (String name : names) {
    System.out.println(name);
}
```

Alhough we are not explicitly invoking *hasNext()* or *next()* methods while iterating over the list, the underlying code that makes this iteration work uses these methods. This implies that the complexity of these operations is hidden from the programmer, but it still exists.

**73. Difference between Stream and Parallel Stream give a real-time example?**

**Sequential Stream:** Sequential Streams are non-parallel streams that use a single thread to process the pipelining. Sequential stream performs operation one by one. Any stream operation in Java, unless explicitly specified as parallel, is processed sequentially. They are basically non-parallel streams used a single thread to process their pipeline.

stream() method returns a sequential stream in Java.

```
class SequentialStreamDemo {

    public static void main(String[] args)
    {
        // create a list
        List<String> list = Arrays.asList( "Hello ",
                                    "G", "E", "E", "K", "S!");

        // we are using stream() method
          // for sequential stream
        // Iterate and print each element
          // of the stream
        list.stream().forEach(System.out::print); //Hello GEEKS!
    }
}
```

In this example the list.stream() works in sequence on a single thread with the print() operation and in the output of the preceding program, the content of the list is printed in an ordered sequence as this is a sequential stream.

**Parallel Streams:** It is a very useful feature of Java to use parallel processing. Using parallel streams, our code gets divide into multiple streams which can be executed parallelly on separate cores of the system and the final result is shown as the combination of all the individual core's outcomes. It is always not necessary that the whole program be parallelized, but at least some parts should be parallelized which handles the stream. The order of execution is not under our control and can give us unpredictably unordered results and like any other parallel programming, they are complex and error-prone.

- ✓ One of the simple ways to obtain a parallel stream is by invoking the **parallelStream()** method of Collection interface.
- ✓ Another way is to invoke the **parallel()** method of Base Stream interface on a sequential stream.

```
class ParallelStreamExample {
    public static void main(String[] args)
    {
        // create a list
        List<String> list = Arrays.asList("Hello ",
                                          "G", "E", "E", "K", "S!");

        // using parallelStream()
        // method for parallel stream
        list.parallelStream().forEach(System.out::print); //ES!KGEHello
    }
}
```

Here we can see the order is not maintained as the list.parallelStream() works parallelly on multiple threads. If we run this code multiple times then we can also see that each time we are getting a different order as output but this parallel stream boosts the performance so the situation where the order is not important is the best technique to use.

**74. Explain the new features of Java 8 Date API?**

Java 8 introduces a new date-time API under the package java.time. Following are some of the important classes introduced in java.time package.

**Local –** Simplified date-time API with no complexity of timezone handling.
**Zoned –** Specialized date-time API to deal with various timezones.

**75. Give examples for LocalDate, LocalTime and LocalDateTime classes?**

LocalDate/LocalTime and LocalDateTime classes simplify the development where timezones are not required.

```java
//Local Date Operations
LocalDate localDate = LocalDate.now();
System.out.println("Local Date is : "+localDate);//2021-04-12

localDate = LocalDate.of(2020, 01, 20);
System.out.println("Local date of : "+ localDate); //2020-01-20

localDate = LocalDate.parse("2020-02-06");
System.out.println("String to Local date conversion "+ localDate);//2020-02-06

//Local Date Operations
LocalTime localTime= LocalTime.now();
System.out.println("Local TIme is : "+localTime);//20:08:14.345938300

localTime = LocalTime.of(12, 30, 55);
System.out.println("Local Time of : "+ localTime);//12:30:55

localTime = LocalTime.parse("12:30");
System.out.println("String to Local date conversion "+ localTime);//12:30

//LocalDateTime Operations
LocalDateTime localDateTime = LocalDateTime.now();
System.out.println(localDateTime);//2021-04-12T20:09:59.496450300
```

**76. How to add the Hours, Minutes, Seconds, Days, Months, Years and Weeks to input date?**

```java
// Local Date Operations
LocalDate localDate = LocalDate.now();
System.out.println("Local Date is : " + localDate);// 2021-04-12
System.out.println("Adding Days : " + localDate.plusDays(2));//2021-04-14
System.out.println("Adding Months : " + localDate.plusMonths(2));//2021-06-12
System.out.println("Adding Weeks : " + localDate.plusWeeks(2));//2021-04-26
System.out.println("Adding Years : " + localDate.plusYears(2));//2023-04-12

// Local Date Operations
LocalTime localTime = LocalTime.now();
System.out.println("Local Time is : " + localTime);//20:17:35.381546200
System.out.println("Adding Hours : " + localTime.plusHours(5));//01:17:35.381546200
System.out.println("Adding Minutes : " + localTime.plusMinutes(20));//20:37:35.381546200
System.out.println("Adding Seconds : " + localTime.plusSeconds(55));//20:18:30.381546200
```

**77. How to format the date/time in java 8?**

Date formatting is supported via two classes mainly i.e. DateTimeFormatterBuilder and DateTimeFormatter.

DateTimeFormatterBuilder works on builder pattern to build custom patterns

DateTimeFormatter provides necessary input in doing so.

```java
//DateTimeFormatter
DateTimeFormatter FOMATTER = DateTimeFormatter.ofPattern("MM/dd/yyyy 'at' hh:mm a");
LocalDateTime localDateTime = LocalDateTime.now();
String ldtString = FOMATTER.format(localDateTime);
System.out.println(ldtString);//04/12/2021 at 08:23 PM

//DateTimeFormatterBuilder
DateTimeFormatterBuilder formatterBuilder = new DateTimeFormatterBuilder();
formatterBuilder.append(DateTimeFormatter.ISO_LOCAL_DATE_TIME).appendZoneOrOffsetId();
DateTimeFormatter formatter = formatterBuilder.toFormatter();
System.out.println(formatter.format(ZonedDateTime.now()));//2021-04-12T20:28:32.9118029Asia/Colombo
```

**78. How to get the Zone specific times (Chicago/Delhi/Chennai etc)?**

```java
LocalDateTime ldt = LocalDateTime.now();
ZonedDateTime klDateTime = ldt.atZone(ZoneId.of("Asia/Kuala_Lumpur"));
System.out.println(klDateTime); //2021-04-12T20:34:00.141037400+08:00[Asia/Kuala_Lumpur]

ZonedDateTime achacago = ldt.atZone(ZoneId.of("America/Chicago"));
System.out.println(achacago); //2021-04-12T20:37:18.066819300-05:00[America/Chicago]

ZonedDateTime akolkata = ldt.atZone(ZoneId.of("Asia/Kolkata"));
System.out.println(akolkata); //2021-04-12T20:37:18.066819300+05:30[Asia/Kolkata]
```

**79. How to find the difference b/w dates, Time, Days, Months and Years?**

```java
public static void main(String[] args) {
    LocalDateTime localDateTime1 = LocalDateTime.of(2000, 01, 03, 11, 52);
    LocalDateTime localDateTime2 = LocalDateTime.of(2021, 07, 06, 10, 22);

    System.out.println("Number of Days between two dates " + ChronoUnit.DAYS.between(localDateTime1, localDateTime2));//7854
    System.out.println("Number of Months between two dates " +ChronoUnit.MONTHS.between(localDateTime1, localDateTime2));//258
    System.out.println("Number of Years between two dates " +ChronoUnit.YEARS.between(localDateTime1, localDateTime2));//21
    System.out.println("Number of Hours between two dates " +ChronoUnit.HOURS.between(localDateTime1, localDateTime2));//188518
}
```

**80. Define Nashorn in Java 8?**

Nashorn is a JavaScript processing engine that is bundled with Java 8. It provides better compliance with ECMA (European Computer Manufacturers Association) normalized JavaScript specifications and better performance at run-time than older versions.

**81. What is the use of Instant class?**

Instant class represents an instant in time to an accuracy of nanoseconds. Operations on an Instant include comparison to another Instant and adding or subtracting a duration.

instant() method of Clock class returns a current instant of Clock object as Instant Class Object. Instant generates a timestamp to represent machine time. So this method generates a timestamp for clock object.

**Ex:**

********

```
Instant instant = Instant.now();
System.out.println(instant.toString());      //2013-05-15T05:20:08.145Z
```

**82. What is the use of Duration class?**

Duration class is a whole new concept brought first time in java language. It represents the time difference between two time stamps.

A Duration object (java.time.Duration) represents a period of time between two Instant objects.

**Ex:**

******

```
Duration duration = Duration.ofMillis(5000);
System.out.println(duration.toString());    //PT5S
```

**83. What is the use of Period class?**

To interact with human, you need to get bigger durations which are presented with Period class.

Java Period class is used to measures time in years, months and days. It inherits the Object class and implements the ChronoPeriod interface.

```
Period period = Period.ofDays(6);
System.out.println(period.toString());   //P6D
```

**84. Explain the JVM changes happened in Java 8?**

**Java 8 JVM Changes:**

**==========**

Before Java 8 Heap area divided into 3 spaces,

1. Younger Generation
2. Older Generation
3. PermGen Memory

In PermGen are we used to store the static content will be stored and it will also stores the application metadata required by the JVM. The biggest disadvantage of PermGen is that it contains a limited size which leads to an OutOfMemoryError. The default size of PermGen memory is 64 MB on 32-bit JVM and 82 MB on the 64-bit version.

Due to the above problems, PermGen has been completely removed in Java 8. In the place of PermGen, a new feature called Meta Space has been introduced. MetaSpace grows automatically by default.

**Metaspace capacity:**

**==========**

By default class metadata allocation is limited by the amount of available native memory (capacity will of course depend if you use a 32-bit JVM vs. 64-bit along with OS virtual memory availability).

A new flag is available (MaxMetaspaceSize), allowing you to limit the amount of native memory used for class metadata. If you don't specify this flag, the Metaspace will dynamically re-size depending of the application demand at runtime.

**Metaspace garbage collection**

**==========**

Garbage collection of the dead classes and classloaders is triggered once the class metadata usage reaches the "MaxMetaspaceSize".

Proper monitoring & tuning of the Metaspace will obviously be required in order to limit the frequency or delay of such garbage collections. Excessive Metaspace garbage collections may be a symptom of classes, classloaders memory leak or inadequate sizing for your application.